INTRODUCCIÓN

El desarrollo de software de un proyecto embebido puede tener muchas aproximaciones, y dependiendo de la complejidad del proyecto es necesario involucrar técnicas que faciliten la migración entre diferentes tecnologías, marcas y plataformas, la reutilización del código y el trabajo en equipo.

Gracias al nivel de integración actual, la disminución de precios y las nuevas tecnologías que permiten tener mayores velocidades de ejecución en los procesadores embebidos, el uso de un sistema operativo de tiempo real es adoptado cada vez más por los grupos de programadores alrededor del mundo.

El uso de un RTOS (*Real Time Operating System*) en el diseño de un sistema embebido tiene gran cantidad de ventajas sobre la forma tradicional de programación, especialmente cuando el grupo de R&D (Research and Development) lo conforman varios diseñadores, porque facilita el mantenimiento del software e independiza a los programadores de la coordinación de varias tareas y del manejo de funciones que dependen del tiempo.

Los sistemas operativos ofrecen un entorno estándar para que el software pueda interactuar con el hardware, suelen ser uniformes, esto les permite funcionar como plataformas en un entorno en el cual las aplicaciones son desarrolladas, de este modo se asegura que una aplicación ejecutada en un ordenador se ejecutará en otro, incluso si éste tiene diferentes recursos.

En este capítulo se muestran dos técnicas que han sido adoptadas por varios programadores: la primera, conocida como loop Consecutivo/Interrupción (*Foreground/Background*), la segunda usa uno de los sistemas operativos más populares llamado el µC/OS-II de la compañía micrium. La primera técnica, aunque no es un sistema operativo como tal, resulta bastante eficiente para procesadores de 8 bits, y se puede complementar con el uso de varios conceptos de RTOS, mientras que la segunda incorpora al proyecto un código muy robusto con posibilidades de expansión y garantía del tiempo real.

Este tema se trata de forma práctica con dos ejemplos reales que muestran su funcionalidad y forma de trabajo dentro del ambiente microcontrolado.

10.1 ¿QUÉ ES UN SISTEMA OPERATIVO DE TIEMPO REAL?

Los sistemas operativos ofrecen un entorno estándar para que el software pueda interactuar con el hardware, suelen ser uniformes, esto les permite funcionar como plataformas en un entorno en el cual las aplicaciones son desarrolladas, de este modo se asegura que una aplicación ejecutada en un ordenador se ejecutará en otro, incluso si éste tiene diferentes recursos.

Es bueno programar como si la máquina receptora fuese de 8 bits, porque para estos procesadores el programador ha de tener en cuenta el tamaño y la velocidad de ejecución que pueden ser críticas;

El RTOS es a su vez un programa que ocupa recursos del procesador, como son espacio en memoria de programa Flash, memoria RAM, base de tiempo (*timer*), y además requiere tiempo para tomar decisiones y manejar las diferentes tareas. Por esta razón, a la familia de procesadores de 8 bits puede afectarles el hecho de incorporar un sistema de estos, y a ellos se dedicará mayor atención en esta sección; una vez el sistema funcione de forma aceptable en 8 bits, su desempeño mejorará en procesadores

superiores como son los de 16 y 32 bits.

En síntesis un sistema operativo de tiempo real RTOS (*Real Time Operating System*), es un programa que coordina el funcionamiento de otros programas llamados tareas. El sistema operativo permite, de forma óptima, el manejo de datos, ejecución de funciones, generación y control de eventos y coordinación general de un sistema.

si luego se pasa a un procesador de 16 o 32 bits el software correrá con mejor desempeño.

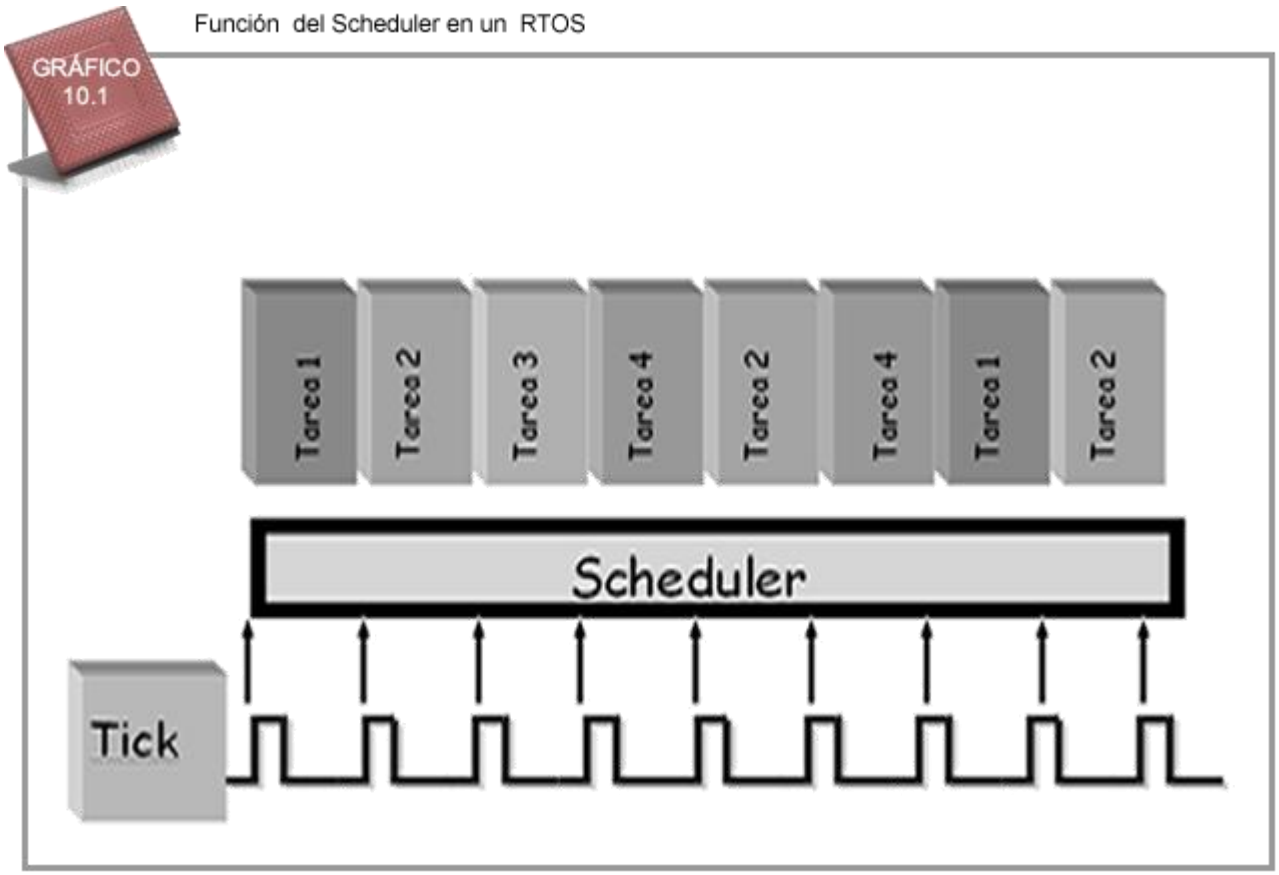
El sistema operativo es principalmente un conjunto de programas que se ejecutan y gestionan el funcionamiento de todos los componentes de la computadora, como el monitor o pantalla, la unidad central de procesamiento, los periféricos de la computadora, el hardware, los controladores y todas las aplicaciones instaladas en el ordenador. Un asistente personal digital, un Smartphone, un teléfono celular, un reproductor de MP3 y otros dispositivos de mano tienen sus propios sistemas operativos y también son la base de computadoras personales o portátiles.

El RTOS tiene la capacidad para adaptarse a las diferentes necesidades y exigencias, puede reconfigurarse o cambiarse totalmente si las tareas son más complejas y adaptarse para llevar a cabo una tarea diferente o cuando aparezcan nuevos programas o hardware que va a ejecutarse en la máquina.

El sistema operativo es un conjunto de programas que ejecutan y gestionan el funcionamiento de todos los componentes de la computadora o máquina.

El RTOS ocupa recursos del procesador para la coordinación de las tareas tanto en tiempo de ejecución como en memoria. El método más común de coordinar la ejecución de las diferentes tareas consiste en incluir en el RTOS un componente llamado el **scheduler,** el cual se encarga de entregar pequeños lapsos de tiempo a cada una de las tareas, suspender su ejecución y pasar el control a la siguiente tarea en lista, de esta forma todas las tareas van ejecutando sus labores.

El tiempo y el cambio de tarea (*context switching*) son gobernados por un reloj externo al RTOS llamado el **TICK**, que consiste en una base de tiempo generada a partir de una interrupción periódica del microcontrolador (*ver Gráfico 10.1*).



De esta forma cada tarea que es suspendida por pequeños lapsos de tiempo, percibe que el procesador esta ejecutándola continuamente, cuando en realidad el procesador está suspendiendo la ejecución de la tarea y regresa.

Los sistemas operativos modernos permiten a la unidad central de procesamiento realizar procesos de multitarea o ejecutar múltiples procesos. El sistema operativo no puede ejecutar al mismo tiempo los procesos, sino que cambia rápidamente de un proceso a otro, ejecutándolos por lo general de acuerdo con las prioridades de funcionamiento del sistema de gestión de los procesos de algoritmo.

10.2 Terminología básica sobre RTOS

10.2.1 Tareas (*Task ó Thread*)

Las tareas son procedimientos formados por códigos de programa que asumen que la CPU está disponible solo para operarla a ella. Las tareas trabajan con base en estados y eventos.

Una tarea es encargada de realizar una labor específica en la aplicación embebida; para su avance, el sistema operativo le entrega pequeñas porciones de tiempo invocando la tarea de forma consecutiva en el punto en el cual la suspendió la vez anterior, dando la impresión de continuidad.

Una tarea en un sistema embebido puede ser la encargada de la inicialización, control y manejo del sistema de visualización LCD, otra tarea podría ser responsable por el manejo del sistema de teclado, otra la encargada de controlar el módulo de comunicaciones seriales con un dispositivo externo, etc.

Prioridad de ejecución de una tarea

Las tareas tienen asociadas ciertas características o atributos, que pueden ser definidas al momento de su creación, o cambiar de forma dinámica a medida que ella se ejecuta, suceden interrupciones o se ejecutan otras tareas.

La prioridad de ejecución se refiere a la importancia que una tarea tiene sobre la ejecución de las demás; esta importancia puede definirse en un sistema con 3 niveles: **BAJA, MEDIA y ALTA**, o bien con un número (ejemplo: entre 0 y 62), que define la prioridad de la tarea, el valor más bajo en número de prioridad, indica mayor importancia; así, una tarea que tenga prioridad 3, tiene mayor importancia para ejecutarse que la que tiene prioridad 10.

La prioridad puede ser fija, definida al principio de la ejecución del sistema, o bien manejarse de forma dinámica en el transcurso del programa.

Dependiendo de la naturaleza del RTOS usado, varias tareas pueden tener en un mismo momento la misma prioridad y en este caso el *scheduler* se encarga de hacer llamado secuencial a cada una de ellas en un esquema denominado **Round-Robin**; sin embargo, algunos sistemas solo permiten una prioridad única por tarea y la que se ejecuta en determinado momento es la que tiene la mayor prioridad, mientras que las demás deben esperar que la tarea de mayor prioridad ejecute su código.

El método más común de coordinar la ejecución de las diferentes tareas consiste en incluir en el RTOS un componente llamado el scheduler, el cual se encarga de entregar pequeños lapsos de tiempo a cada una de las tareas para crear el efecto de simultaneidad.

El sistema operativo, no puede ejecutar al mismo tiempo los procesos, sino que cambia rápidamente de un proceso a otro, ejecutándolos de acuerdo con las prioridades de funcionamiento diseñadas por el programador.

La prioridad de ejecución se

refiere a la importancia que una tarea tiene sobre la ejecución de las demás; esta importancia puede definirse en un sistema con 3 niveles: BAJA, MEDIA y ALTA, o bien con un número (ejemplo: entre 0 y 62).

Una tarea puede variar su prioridad a medida que corre el programa; por ejemplo, el manejo del display puede ser una tarea de prioridad BAJA si el teléfono está inactivo, pero puede pasar a ser de prioridad ALTA cuando se tiene una llamada activa.

Pero la prioridad puede ser cambiada de forma dinámica, y esto sí lo permite la gran mayoría de los RTOS comerciales. En un sistema multitarea que administra un equipo celular, puede ser de prioridad ALTA el manejo de la batería cuando su nivel está por debajo de un nivel crítico. Sin embargo, puede pasar a ser una tarea de prioridad MEDIA si el nivel esta en un rango seguro y tal vez pasar a prioridad BAJA si el equipo está conectado a la red eléctrica.

El manejo del display puede ser una tarea de prioridad BAJA si el teléfono está inactivo, pero puede pasar a ser de prioridad ALTA cuando se tiene una llamada activa, realiza la reproducción de un archivo de video o una teleconferencia.

Estados de tarea

Una tarea puede encontrarse en diferentes situaciones, dependiendo de su estado en ejecución (*Gráfico 10.2*):

Suspendida (*Dormant***)**: indica que una tarea está creada o terminada y no está siendo ejecutada por el RTOS ni programada su ejecución por el *scheduler*.

Ejecución (*Run***):** una tarea se encuentra en este estado cuando el scheduler del RTOS ha pasado el control a su código y está siendo ejecutada en el presente por la CPU.

Disponible (*Ready***):** una tarea que está esperando turno para ser ejecutada por el scheduler. Su prioridad es más baja que la que está en modo **Run**.

Espera (*Wait***):** estado en el que una tarea está a la espera de algún evento para continuar su ejecución, hasta que no se presente dicho evento no estará en la lista de tareas que son ejecutadas por el *scheduler*. El evento que espera la tarea puede ser el cambio de un pin externo, que un recurso esté disponible, que pase un determinado tiempo, que algún proceso de otra tarea termine, etc.

Interrumpida (*Interrupted***):** cuando una tarea estando en modo Run, es suspendida por la aparición de una **ISR**. En este caso el **RTOS** no tiene participación en la administración de suspender y retornar el control a la tarea.

Estado básico de las tareas en un RTOS[1].

1 Gráfico cortesía **www.micrium.com**

10.2.2 Recursos (*resources*)

Los recursos compartidos con exclusión mutua no permiten

ser accesados por dos tareas de forma simultánea; ejemplo de ello es un disco duro para almacenamiento de información, un display de visualización de texto o gráficos o una impresora externa.

Los recursos son entidades usadas por una tarea. Un recurso puede ser una función no reentrante, una estructura, una variable, un periférico interno del microcontrolador, un dispositivo externo como una memoria serial, un display, un teclado, una impresora, un sistema de almacenamiento, etc. Se dice que un recurso es compartido (*shared resource*), cuando es usado por más de una tarea, debido a que cada tarea puede tener acceso exclusivo al recurso en un determinado espacio de tiempo, y prevenir que otra tarea lo haga, con el objetivo de prevenir daños en los datos o en los dispositivos, este proceso es denominado exclusión mutua *(mutual exclusión)*. Casos típicos de recursos compartidos con exclusión mutua son un disco duro para almacenamiento de información, un display de visualización de texto o gráficos, una impresora externa, entre otros.

10.2.3 Eventos (*events*)

Son elementos de notificación hacia una tarea para indicar la ocurrencia de un hecho o estímulo que interesa a una o varias tareas que se están ejecutando. Los eventos pueden llegar ya sea del hardware interno o externo al procesador, o bien pueden ser eventos de software: *timeouts*, interrupciones o un resultado de la operación de una función.

10.2.4 Semáforos (semaphores)

Los semáforos son mecanismos de control que proveen al RTOS una forma de prevenir que múltiples tareas realicen acceso al mismo recurso en el mismo tiempo. Se usan además para indicar la ocurrencia de un evento, y permiten sincronizar varias actividades dentro del sistema. Una vez una tarea va a iniciar un proceso con un recurso, pregunta por el estado del semáforo; si el semáforo está activo la tarea quedará en el estado de *wait*, si no está activo, lo toma y lo bloquea, realiza su operación en el recurso y una vez termina, libera el semáforo, permitiendo que otras tareas a partir de ese momento puedan hacer uso del recurso de la misma forma.

10.2.5 Mensajes (*message mailbox*)

Un mensaje es un objeto que se entrega a una tarea, o bien puede ser una ISR[2], un apuntador con determinada medida a otra tarea. Este apuntador normalmente esta inicializado a una estructura de datos que contiene un mensaje. Existen en el sistema operativo funciones ya incorporadas que permiten crear, dejar el mensaje pendiente, aceptar, preguntar si existe un mensaje específico para alguna tarea.

2 ISR: Interrupt Service Routine.

10.2.6 Bloques de Memoria (*buffers*)

Son elementos de almacenamiento continuos para la capa de aplicación, permite crear listas circulares, FIFO[3] o LIFO[4] de cualquier tamaño. Las listas pueden crearse y borrarse de forma dinámica, permitiendo de esta forma un manejo eficiente de la memoria. Funcionan de forma similar a las funciones *malloc()* y *free()* para el manejo dinámico de memoria.

---

3 FIFO: First Input First Output.

4 LIFO: Last Input First Output.

10.2.7 Reloj y Timers (*Clock Tick & Timers*)

El "clock tick" o tick es una interrupción que ocurre de forma periódica, le permite al RTOS contar el tiempo que entrega a cada tarea y el manejo de timeouts.

El valor del tick de frecuencia más alta genera mayor preámbulo *(overhead)* en el sistema y tiene un mayor consumo de energía, debido a que debe tomar decisiones más veces por segundo, lo cual puede ser crítico para el desempeño de la máquina; sin embargo, un tiempo muy largo de *tick* ocasiona que las tareas tengan tiempo de respuesta más lentos y que la sensación de "tiempo real" se vea afectada.

El valor del tick le permite al diseñador controlar los requerimientos de velocidad, consumo y tiempo de respuesta del sistema.

Es responsabilidad del diseñador la selección adecuada del valor del tick dependiendo de los requerimientos de velocidad, consumo y tiempo de respuesta requerido. Valores típicos para un sistema embebido pueden ir desde 10 mseg a 100 mseg, dependiendo de los requerimientos de una aplicación particular.

10.2.8 *Kernel*

El *kernel* es la parte del RTOS responsable por la distribución del tiempo de la CPU para el manejo de las tareas, además de la comunicación entre ellas. El servicio fundamental del *kernel* es el **cambio de contexto** (véase Capítulo 1.6). El uso de un *kernel* de tiempo real generalmente simplifica el diseño de sistemas permitiendo que la aplicación pueda ser dividida en múltiples tareas que el *kernel* administra una a una.

La parte del *kernel* que se encarga de determinar cual tarea debe ejecutarse es el scheduler, el que basado en la prioridad, pasa el control de la CPU.

Como es sabido, el cambio de contexto agrega preámbulo, debido a que el *kernel* requiere también espacio de tiempo y CPU para realizar el cambio entre tareas, este tiempo debe ser contabilizado y ha de ser insignificante para la aplicación.

Es de esperarse también que para procesadores de 8 bits este porcentaje sea mayor, por las limitaciones de la arquitectura, pero no debería superar más del 5%, en arquitecturas de 16 y 32 bits, este porcentaje no sobrepasa el 2% lo que indica que el sistema operativo será más eficiente en máquinas más robustas.

Cuando se cambia de una tarea a otra, es decir, se produce un cambio de contexto, se agrega un preámbulo, debido a que el kernel requiere también espacio de tiempo y CPU para realizar el cambio entre tareas, y dicho lapso debe ser contabilizado y ha de ser insignificante para la aplicación.

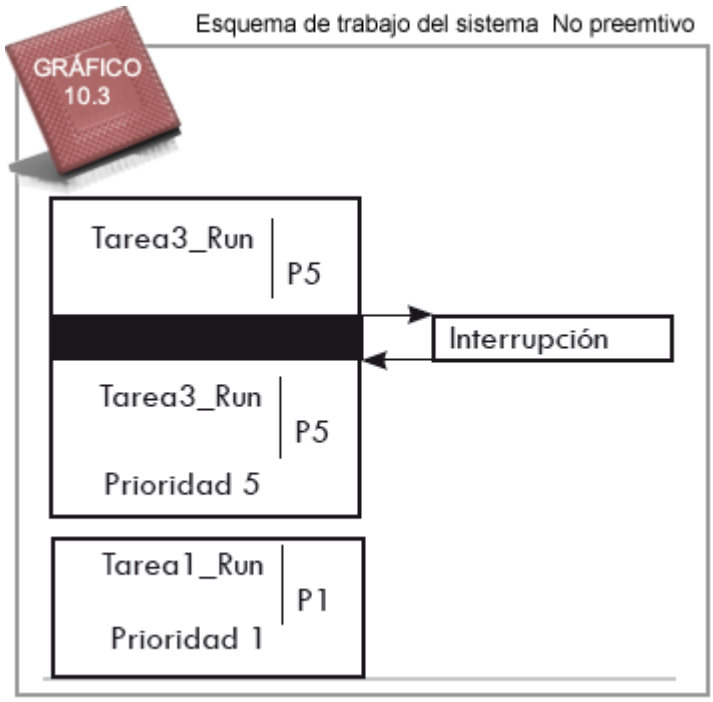
En un sistema No preemtivo el kernel no suspende la ejecución de la tarea actual, sino que las tareas entregan el control al sistema de forma voluntaria; sin embargo, las interrupciones pueden quitar el control en cualquier momento.

kernel No preemtivo

Un sistema No preemtivo no suspende la tarea que está en estado **Run**, por lo que este tipo de *kernel* es también llamado un sistema cooperativo, porque las tareas entregan el control al sistema de forma voluntaria, sin embargo las interrupciones pueden quitar el control en cualquier momento.

Para mantener la sensación de continuidad el proceso de entregar el control de nuevo a la tarea debe ser frecuente.

En el *Gráfico 10.3* una vez el *scheduler* toma la decisión de pasar el control a la Tarea3 por ser la de mayor prioridad en el momento, ésta inicia su ejecución, en el transcurso sucede una interrupción que le cambia a la Tarea1 su prioridad a 1, la cual es mayor que la que tiene la Tarea3 (prioridad 5); una vez la interrupción retorna, el control se pasa a la Tarea3 (siendo de menor prioridad), y es ésta la que voluntariamente entrega el control al *scheduler*, el cual pasa luego el control a la Tarea 1, que ahora es la de mayor prioridad.



Una de las ventajas del sistema No preemtivo es que las interrupciones carecen de preámbulo, debido a que no tienen que tomar decisión alguna sobre a cuál tarea entregarle el control, sin embargo, tiene la desventaja que el tiempo de respuesta de las tareas puede ser un poco más lento, debido a que la tarea que está pendiente de un evento, debe esperar que la tarea de menor prioridad entregue el control y siga su turno, además, el tiempo de respuesta es no determinístico debido a que no se sabe a ciencia cierta cuanto tiempo tardará en entregarse el control. De ello se deduce la importancia que todas las tareas realicen procesos muy rápidos en modo Run, esto evitará que otras tareas más importantes se atrasen.

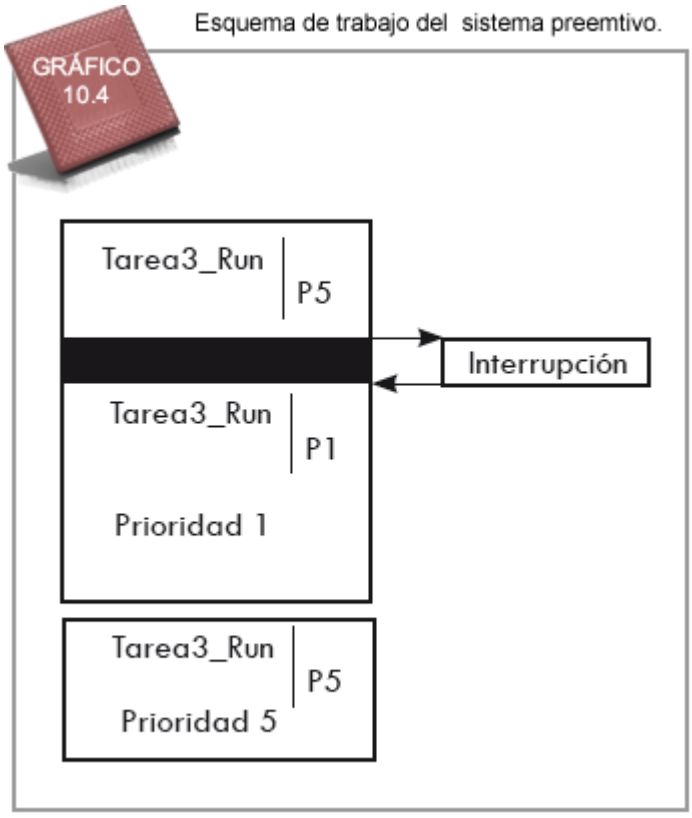
Es importante que todas las tareas realicen procesos muy rápidos en modo Run, esto evitará que otras tareas más importantes se atrasen.

kernel preemtivo

Un *kernel* preemtivo es aquel que suspende la tarea en estado **Run** y le pasa el control a una de mayor prioridad. El *scheduler* en su labor entrega el control a la tarea de mayor prioridad, esta inicia su ejecución, pero en algún momento de su ejecución, sucede una interrupción que genera el cambio de prioridad o entrega un evento a una tarea de máxima prioridad, en esta caso se suspende la tarea a la que se entregó el control y se pasa el control a esta nueva tarea de máxima prioridad.

Un kernel preemtivo suspende la tarea en estado Run y le pasa el control a una de mayor prioridad.

Este esquema de trabajo se ilustra en el *Gráfico 10.4.* Inicialmente el *scheduler* para el control a la Tarea3 por ser la de mayor prioridad, mientras está ejecutando esta tarea se presenta una interrupción que cambia la prioridad de la Tarea1 a 1, siendo ésta una prioridad mayor a la de la Tarea3, en este esquema, el control no se pasa a la Tarea3 en estado interrumpida, sino que lo pasa a la Tarea1 por ser ahora de mayor prioridad.



Este esquema tiene la ventaja de garantizar un poco más el "tiempo real" de las tareas de alta prioridad, debido a que la ejecución de la tarea de mayor prioridad es determinístico (se puede medir); además, permite disminuir el tiempo de respuesta de la tareas a un nivel inferior.

Las aplicaciones que usan *kernel* preemtivo tienen más control y requieren mayor manejo de semáforos, debido a que puede darse más frecuente el caso que dos (2) tareas traten de usar el mismo recurso al tiempo.

10.3 SISTEMA DE LOOP CONSECUTIVO/INTERRUPCIÓN

Llamado el sistema *Foreground/Background*, es una técnica bastante usada en sistemas de baja complejidad o de limitación de procesamiento, debido a que no tiene preámbulo (*overhead*) entre cambio de contexto de una tarea a otra.

El cambio se realiza por medio de las instrucciones nativas del procesador, JSR (*jump to subroutine*) o CALL, para entregar el control a una tarea y ponerla en modo **Run** y RTS (*Return From Subroutine*), o RTC (*Return From Call*), para ponerla en estado *Wait* o **Ready**, y continuar con la ejecución de la siguiente tarea en la cola.

10.3.1 Diseño del Loop principal

El diseño del módulo principal consiste en generar un loop infinito en el main (*Gráfico 10.5*) y realizar llamado consecutivo a todas las tareas (*background*) que el sistema requiere procesar, las interrupciones generales se habilitan una vez que los módulo son inicializados, siempre y cuando no se requiera el suceso de interrupciones en los módulo de inicialización permitiendo que interrumpan la ejecución de una tarea; solo se deshabilitan por pequeños lapsos de tiempo en las zonas críticas del programa.

Programa principal esquema Loop Consecutivo/Interrupción.

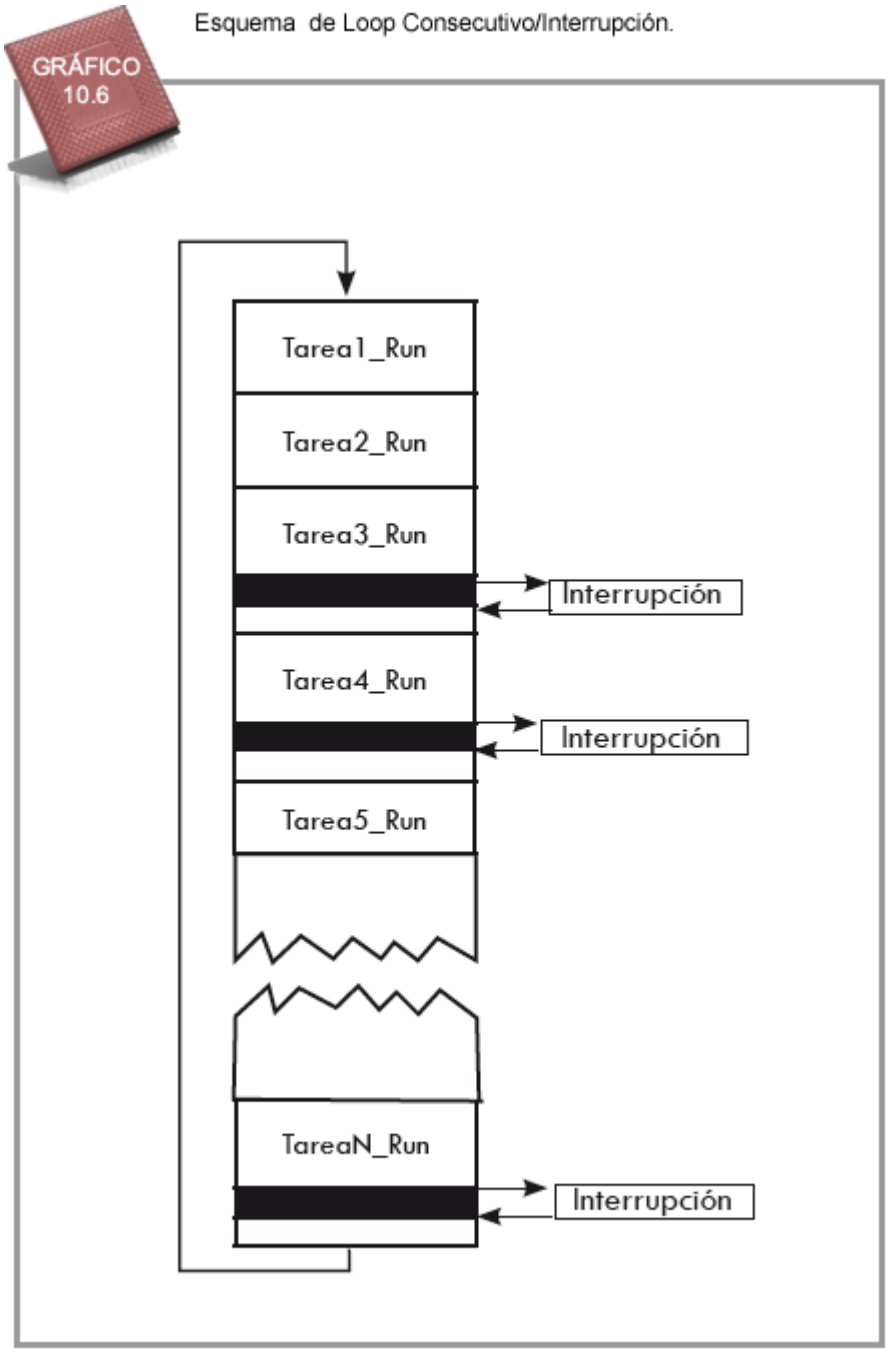
```
main(void){
        Mcu_Init();                 //inicialización CPU y MCU
        Tarea1_Init();  //inicialización tareas
        Tarea2_Init();
        Tarea3_Init();
        Tarea4_Init();
        Tarea5_Init();
        ...
        TareaN_Init();
                                    //habilita interrupciones
        EnableInterrupts;

for(;;){            //loop infinito
        Tarea1_Run();
        Tarea2_Run();
        Tarea3_Run();
        Tarea4_Run();
        Tarea5_Run();
        ...
        TareaN_Run();
}
}
// fin del módulo principal
```

Las interrupciones se encargan de manejar los eventos asincrónicos (*Foreground*).

El sistema completo consta de 2 niveles de programa: el nivel de tareas (*Task Level*) y el nivel de interrupciones (*Interrupt Level*), como se puede ver en el *Gráfico 10.6.*

Esquema de Loop Consecutivo/Interrupción.



En este esquema las operaciones críticas deben ser realizadas por las interrupciones (**ISRs**) para garantizar que los procesos cumplen con los tiempos requeridos. Debido a esto las ISRs tienden a ser un poco más largas de lo que deberían, además que la información para el nivel de tareas no es procesada hasta que llegue su turno de ejecución en el loop consecutivo; esta demora es llamada tiempo de respuesta del nivel de tareas (*Task-Level response*), este tiempo depende de que tanto tiempo tarde el Loop consecutivo en ejecutarse, y puede no ser constante, debido a que depende del procesamiento